Colecciones
o
Estructuras de Datos

1. Introducción
2. Colecciones
3. La interfaz Iterator
4. Set
5. List
6. Maps
7. La clase Collections

En Java se puede trabajar con estructuras de datos clásicas (Objetos) o bien con sus definiciones.

- Hasta la versión 1.1:
  - Existían las clases **Vector, Stack y Hashtable**
    - » todos sus métodos están **sincronizados** -> implica rendimiento muchísimo menor
    - » ahora para sincronizar -> utilizar **Collections.sincronizedX()**

Pero existen todo un conjunto de clases que nos facilitan el trabajo:

- Desde la versión 1.2:
  - **Iterator** -> para iterar
  - **Collection** -> contenedores de objetos
    - » **List** -> prima orden, se permiten duplicados
    - » **Set** -> no permiten duplicados (sobreescribir **equals() y hashCode()**)
  - **Map** -> asociaciones clave/valor
- A parir de la versión 5.0:
  - **Genéricos** (parametrización)
  - **for-each** (Interfaz **Iterable**)

2.- Colecciones

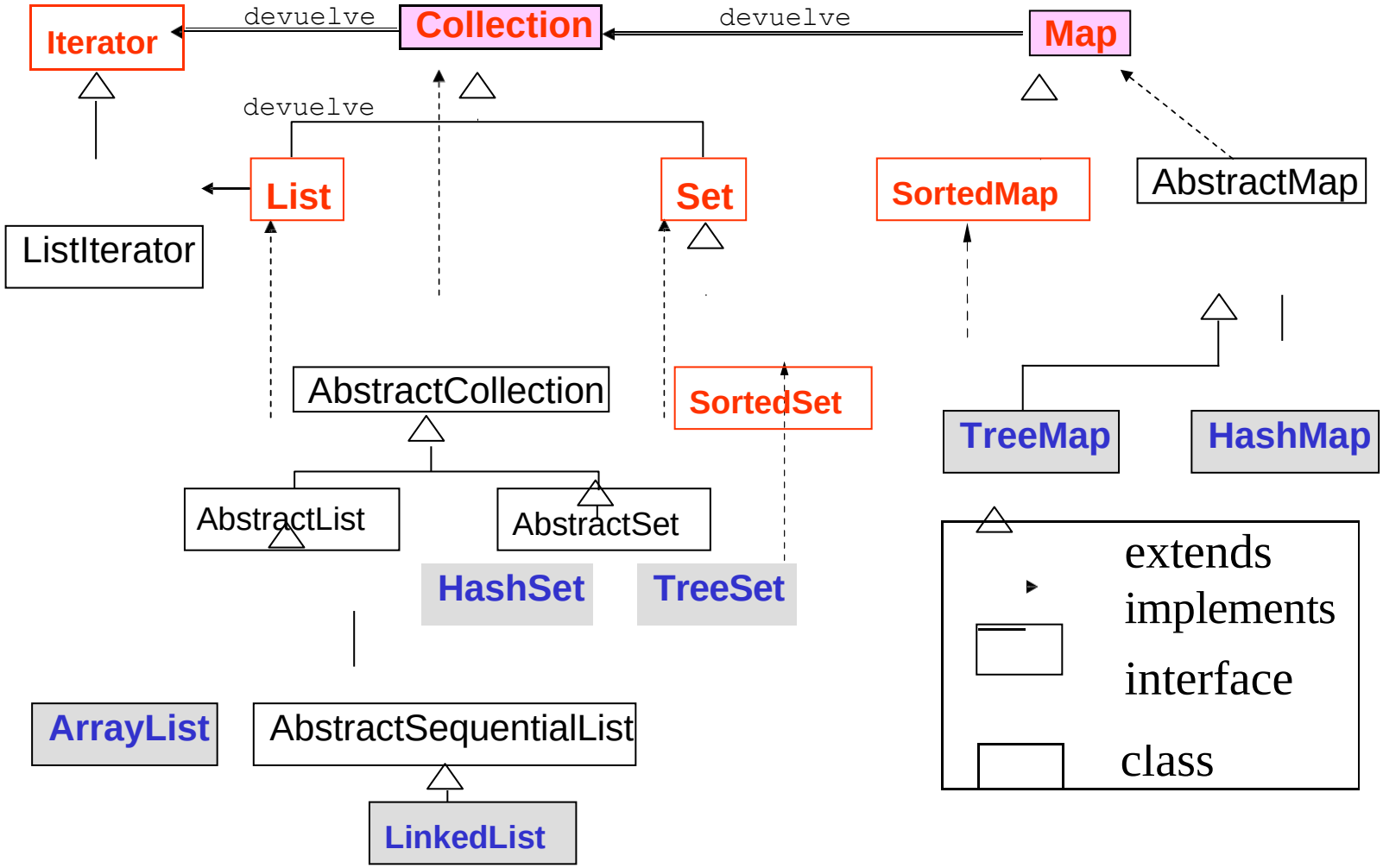
- Muchos programas requieren de mantener un conjunto de datos relacionados.
- Una colección. Es un objeto que agrupa múltiples elementos.
- Las colecciones son utilizadas para almacenar, obtener, manipular y comunicar datos que han sido agregados.
- En su gran mayoría están contenidas en el paquete java.util

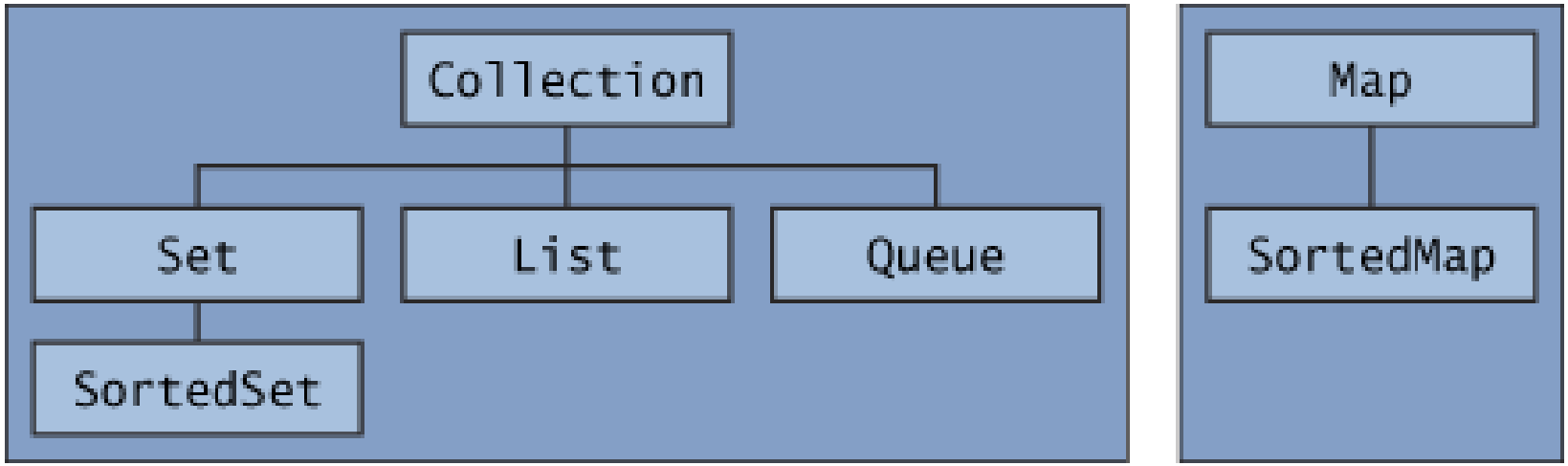
2.1 Beneficios del uso de Colecciones

- Reducen el esfuerzo de programación.
- Incrementan la velocidad y calidad al programar.
- Permiten mantener interoperatividad entre API de diferentes distribuidores.
- Reducen el esfuerzo de aprender nuevos API
- Reducen el esfuerzo de diseñar un nuevo API
- Fomentan el reutilización de software

2.2.-Colecciones en Java

- Las colecciones en Java son un **ejemplo** destacado de implementación de **código reutilizable** utilizando un lenguaje orientado a objetos.
- Todas las colecciones son **genéricas**.
- Los tipos abstractos de datos se definen como **interfaces**.
- Se implementan **clases abstractas** que permiten factorizar el comportamiento común a varias implementaciones.
- Un mismo **TAD** puede ser implementado por varias clases
  - -ej: **`List`**: **`LinkedList`**, **`ArrayList`**


Iterator
devuelve
Collection
devuelve
Map
devuelve
List
Set
SortedMap
AbstractMap
ListIterator
AbstractCollection
SortedSet
TreeMap
HashMap
AbstractList
AbstractSet
HashSet
TreeSet
ArrayList
AbstractSequentialList
LinkedList
extends
implements
interface
class


Collection
Set
List
Queue
SortedSet
Map
SortedMap

2.3.-La interfaz Collection

- Una Collection representa un grupo de objetos conocidos como elementos. La interfaz Collection es usada para agrupar objetos de la manera más general posible.
- Define las operaciones comunes a todas las colecciones de Java.
- Permite usar colecciones basándonos en su interfaz en lugar de en la implementación.
- Los tipos básicos de colecciones son (subtipos de `Collection<T>`) :
  - Listas, definidas en la interfaz `List<T>`
  - Conjuntos, definidos en la interfaz `Set<T>`
- El uso de las Colecciones se basa en utilizar interfaces y seleccionar la Collection que esté acorde a nuestras necesidades.

2.4.- Uso de la interfaz Collections

- Ejemplo. Agregar a una Collection:

```
//Método
Collection resultado = new ArrayList();
ListaDTO registro = null;
Connection conexion = abrirConexion();
try {
    String sql = "SELECT * FROM listas ";
    Statement sentencia =
              conexion.createStatement();

    ResultSet rs = sentencia.executeQuery(sql);
    while (rs.next()) {
        resultado.add(crearDTO(rs));
    }
} //Continuación del método
```

Interfaz `Collection<T>`

Operaciones básicas de consulta:

- **`size`**`()`: devuelve el número de elementos.
- **`isEmpty`**`()`**:** indica si tiene elementos.
- **`contains`**`(Object e)`**:** indica si contiene el objeto pasado como parámetro utilizando el método **`equals`**.

Operaciones básicas de modificación:

- **`add`**`(T e)`**:** añade un elemento a la colección.
  - Retorna un booleano indicando si acepta la inserción.
- **`remove`**`(Object e)`**:** intenta eliminar el elemento.
  - Retorna un booleano indicando si ha sido eliminado.
  - Utiliza el método **`equals`** para localizar el objeto.
- **`clear`**`()`**:** elimina todos los elementos.
- **`addAll`**`(Collection<? extends T> col)`**:** añade todos los elementos de la colección `col`
- **`removeAll`**`(Collection<?> col)`**:** elimina todos los objetos contenidos en `col`

3.- La interfaz Iterator

- La iteración de la Collection consiste en utilizar un objeto que implemente la interfaz Iterator, con el objeto de recorrer sus elementos.
- **Interfaz Iterator**
  - Desde la versión 5.0 todas los contenedores implementan la **interfaz Iterable**.
  - Nos sirve para iterar con el **for-each**.
  - Funcionamiento:

Uso de Collections

- Ejemplo. Recorrer a una Collection:

```
// Implementación Método
//Resultado representa la Collection del
//ejemplo anterior.
Iterator iterator = resultado.iterator();
while (iterator.hasNext()) {
     RegistroDTO dto = (RegistroDTO)iterator.next();
     System.out.println("Registro: "+dto);
}
//Continuación del método
```

3.2.-Uso iterator con Collections JDK 5

- Ejemplo. Recorrer a una Collection en JDK1.5:

```
// Implementación Método
//Resultado representa la Collection del
//ejemplo anterior.
//Collection<RegistroDTO> resultado = new
//      ArrayList<RegistroDTO>();

for (RegistroDTO dto : resultado ) {
     System.out.println("Registro: "+dto);
}
//Continuación del método
```

La interfaz List

- Una lista es una colección ordenada (En algunas veces se llama secuencia).
- Una lista puede contener elementos duplicados.
- Mantiene las operaciones heredados en la interfase Collection

La interfaz List

□Las clases que implementan la interfaz List son:

- ArrayList: Un arreglo dinámico y modificable en tamaño.
- LinkedList: Cada uno de los elementos contiene un apuntador al elemento anterior y al siguiente.
- Vector: Pertenece a Java desde la primera distribución e implementa una lista utilizando arreglos (obsoleto).

Interfaz `List<T>`

- La interfaz **`List<T>`** define secuencias de elementos a los que se puede acceder atendiendo a su posición.

- Las posiciones van de `0` a `size()-1`.
  - El acceso a una posición ilegal produce la excepción `IndexOutOfBoundsException`
- El método **`add`**`(T e)` añade al final de la lista.
- Añade a las operaciones de `Collection` métodos de acceso por posición como:
  - `T` **`get`** `(int index)`
  - `T` **`set`** `(int index, T element)`
  - `void` **`add`** `(int index, T element)`
  - `T` **`remove`** `(int index)`

- ArrayList<T>
  - Implementación basada en arrays redimiensionables.
  - Operaciones de inserción y modificación ineficientes.
  - Operaciones de creación y consulta rápidas.

- LinkedList<T>
  - Implementación basada en listas doblemente enlazadas
  - Inserciones y modificaciones rápidas, especialmente en el principio y el final:
    - Métodos no disponibles en `List<T>`:
      `addFirst, addLast, removeFirst, removeLast`
  - Acceso aleatorio a elementos ineficiente.
  - Acceso eficiente al principio y al final de la lista:
    - `getFirst` y `getLast`

La interfaz Set

- La interfase Set es una colección que no permite elementos duplicados.

- La interfase Set tiene únicamente los métodos de la interfaz Collection y sólo le agrega la característica de que no pueden existir elementos repetidos.

La interfaz Set

- Las clases que implementan a la interfaz Set son:
  - HashSet: No permite elementos duplicados y utiliza el hashing para almacenar los elementos.
  - TreeSet: Ordena los elementos utilizando un árbol binario.

Interfaz `Set<T>`

- La interfaz **`Set<T>`** define conjuntos de elementos no repetidos.
- Implementaciones de conjuntos:
  - **`HashSet<T>:`**
    - Guarda los elementos del conjunto en una tabla *hash*.
    - Para evitar la inserción de elementos repetidos, la igualdad de los objetos se comprueba comparando los **`hashCode`**, si son iguales se compara con **`equals`**.
  - **`TreeSet<T>:`**
    - Implementación de **conjuntos ordenados** basada en árboles binarios balanceados.
    - Para su funcionamiento es necesario definir un **orden** (se estudia más adelante).
- Las operaciones de búsqueda y modificación son más lentas en `TreeSet` que en `HashSet`

La interface Map

- Los mapas almacenan objetos basados en llaves únicas.
- Los mapas pueden soportar elementos repetidos, pero no llaves repetidas.
- Esta clase no extiende de la interfase Collection.
  - Define un contenedor de **asociaciones clave/valor**.
  - **NO permite claves duplicados**
  - Si modificamos el estado de una clave, su comportamiento es impredecible, por lo que se recomienda utilizar **objetos inmutables** (no cambian su estado) para las claves.
  - Los claves deberían sobrescribir los métodos **hasCode() y equals()**
    - Su implementación más importante es **HashMap**
    - Hereda de ella la Interfaz **SortedMap**:
      - Las claves están **ordenadas**, por ello:
        - hay que utilizar un **Comparator**.
        - o que las claves implementen la interfaz **Comparable**.
      - Su implementación más importante es **TreeMap**

La interface Map

- Las clases que implementan la interfaz Map son:
  - HashMap/ HashTable: Usa el algoritmo de hashing para almacenar los elementos
  - TreeMap: Provee un Mapa ordenado.

Otras características

- Los objetos de tipo Collection o Map son contenedores, que a diferencia de las matrices, incrementan su capacidad cuando lo necesitan.
- **loadFactor** = size / capacity
- Si size > loadFactor
  - □-> se incrementa la capacidad
  - □-> se crea una nueva estructura de datos
  - □-> se copia los elementos de una a otra
- Para evitar ampliaciones sucesivas **initialCapacity** debería ser lo más cercano al tamaño esperado.
- Las clases **Collections** y **Arrays** son clases de utilidades.

Novedades Java 7: InnerClass

```
7  public class Principal2 {
8
9      public static void main(String[] args) {
10
11         Persona personaA = new Persona("pedro", "perez", 20);
12         Persona personaB = new Persona("ana", "blanco", 15);
13         Persona personaC = new Persona("miguel", "alvarez", 50);
14
15         List<Persona> lista = Arrays.asList(personaA, personaB, perso
16
17         lista.sort(new Comparator<Persona>() {
18
19             @Override
20             public int compare(Persona p1, Persona p2) {
21
22                 return p1.getEdad() < p2.getEdad() ? 1 : -1;
23             }
24
25         });
26         for (Persona p : lista) {
27
28             System.out.println(p.getNombre() + " ," + p.getEdad());
29
30         }
31
32     }
```

Novedades en Java 8

Lambda Expressions

Una expresión lambda se compone de dos elementos. En primer lugar de un conjunto de parámetros y en segundo lugar de una expresión que opera con los parámetros indicados.